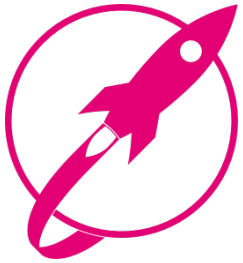


FH·W-S

iOS Programmierung

(mit Swift)

Peter Braun, Florian Bachmann & Andreas Wittmann
@pe_braun @florianbachmann @anwittmann

Deutsche Telekom AG
FHWS - Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt

#FHWSSwift

Agenda

1. **Introduction** – Organisatorisches
2. **First iOS-Project** – Hello World, **First iOS-Project** – Still Hello World (now with Code 😄)
3. **Swift**, Wait!, What about Objective-C?, Why Swift?
4. **A (not so) Quick Tour**
5. **Documentation**
6. **The basics –** iOS Architecture & more
7. **User Interfaces –** View Controller, Auto Layout & Size Classes
8. **Storyboard & Segues**
9. **Tables & NavigationController**
10. **TabBarController**
11. **Notifications**
12. **PickerViews**
13. **Touches, Gestures, 3D Touch, Peek & Pop**
14. **ScrollView & StackViews**
15. **Networking** – JSON & Dependency Managers
16. **WebKit**
17. **Maps**
18. **Storage & Data persistency –** NSUserDefaults, NSKeyedArchiver & Core Data
19. **ObjC**

Swift
A (not so) Quick Tour

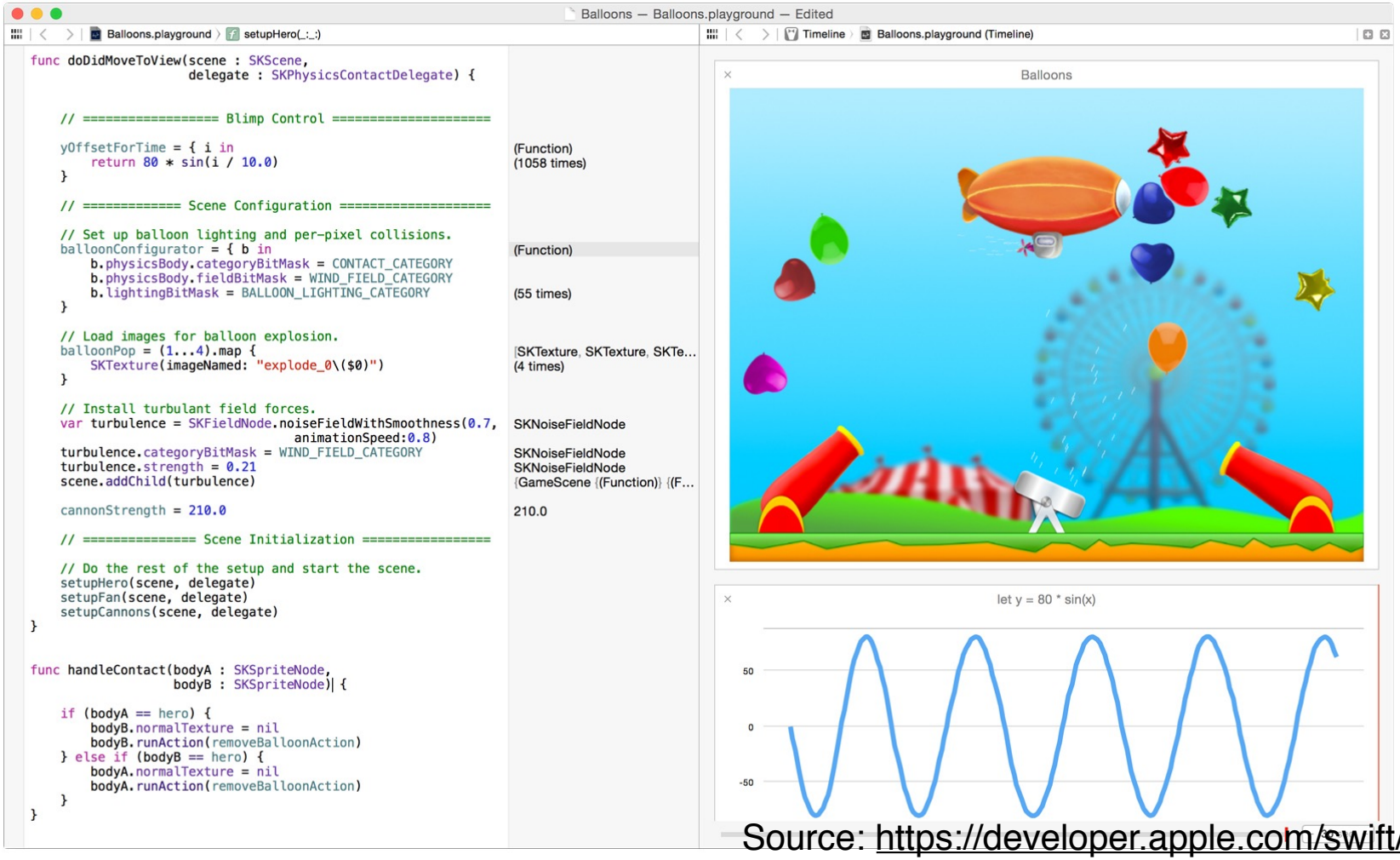
```
print("Hello Würzburg! We ❤️ Swift!")
```

Swift - A (not so) Quick Tour

we will see:

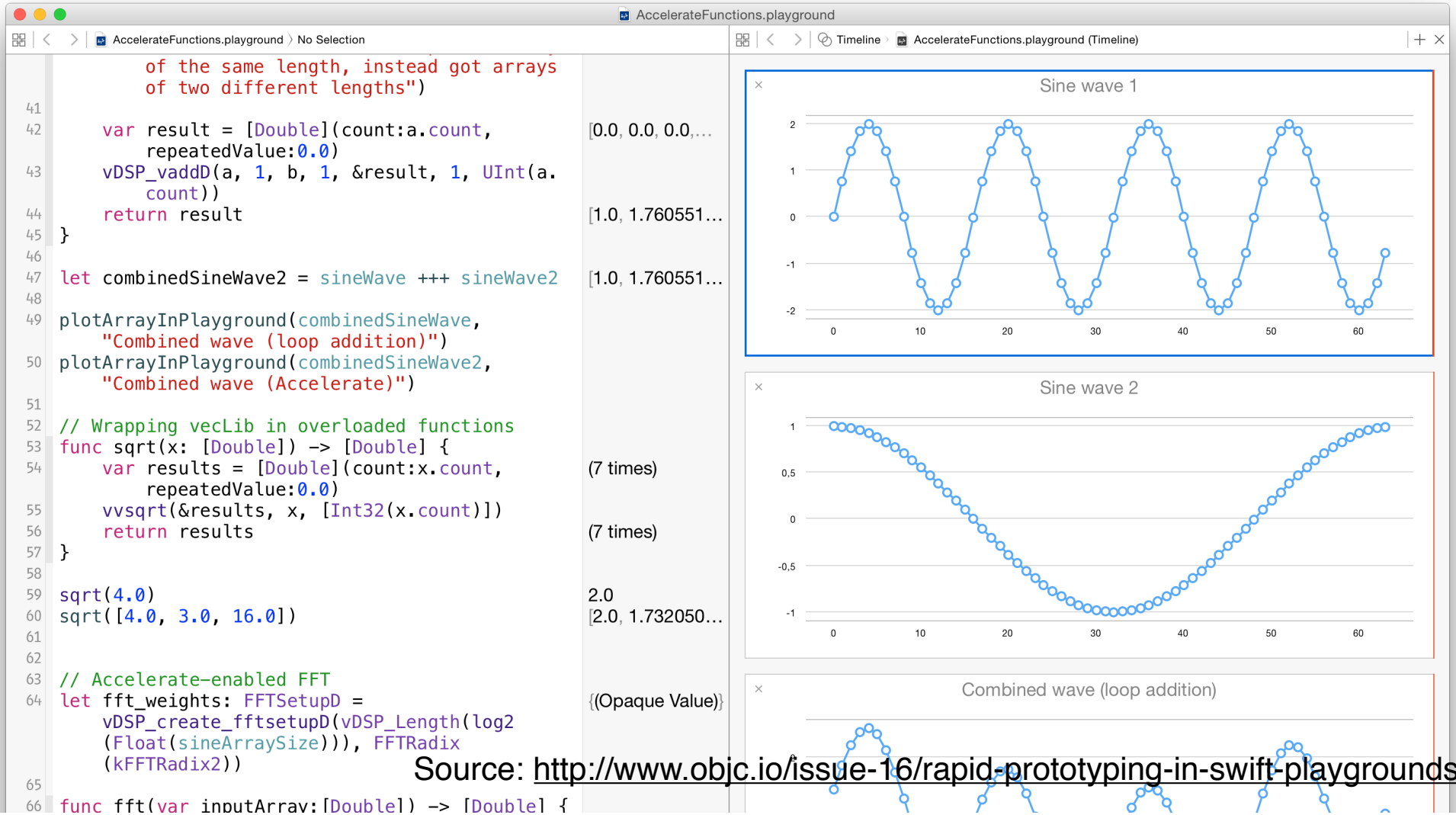
1. The Basics
2. Basic Operators
3. Strings and Characters
4. Collection Types
5. Control Flow
6. Functions
7. Closures
8. Enumerations
9. Classes and Structures
10. Properties
11. Methods
12. Subscripts
13. Inheritance
14. Initialization
15. Deinitialization
16. Automatic Reference Counting
17. Optional Chaining
18. Error Handling
19. Type Casting
20. (Nested Types)
21. Protocols
22. Extensions
23. Generics
24. (Access Control)
25. (Advanced Operators)
26. Swift Standard Library

Playgrounds (1/2)

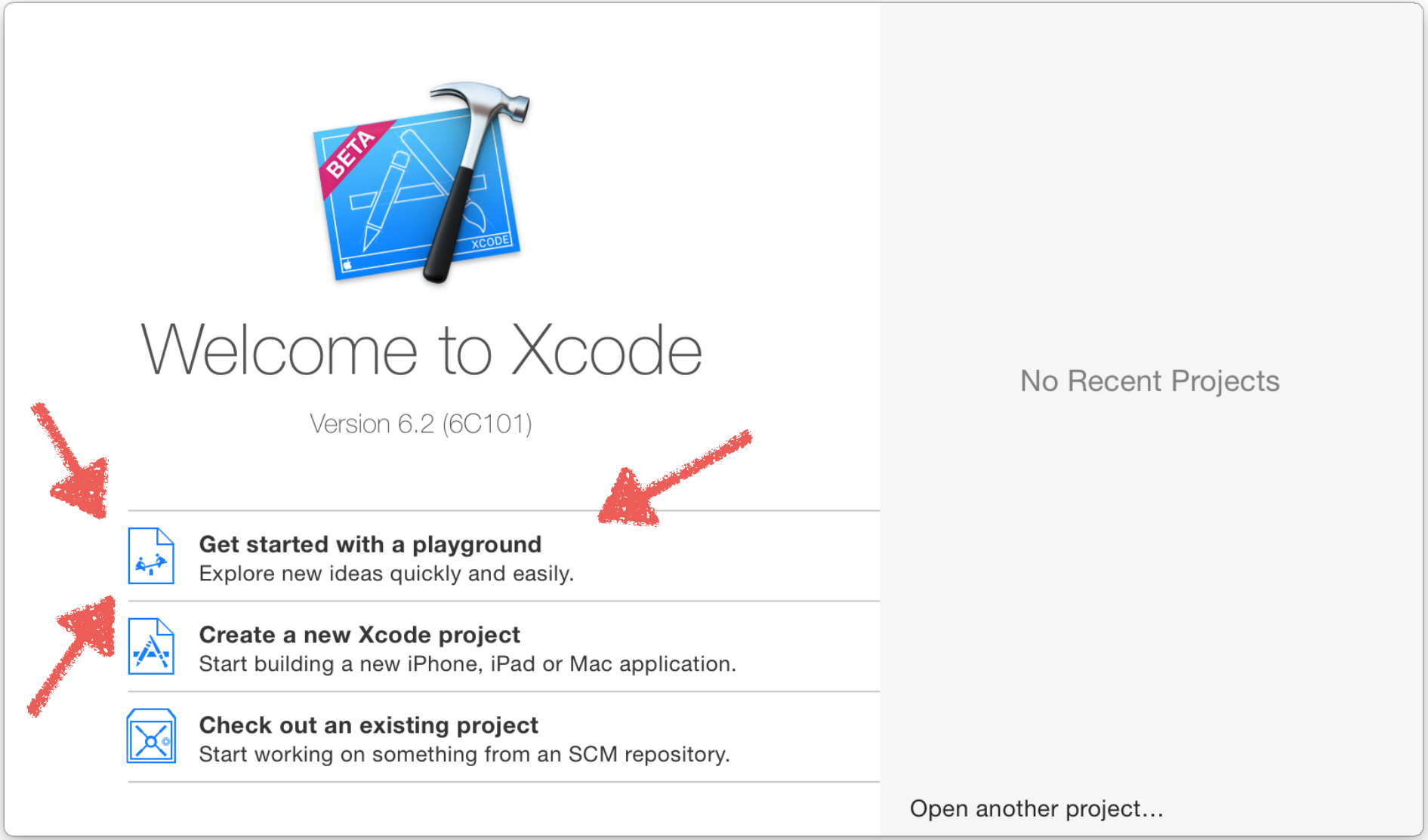


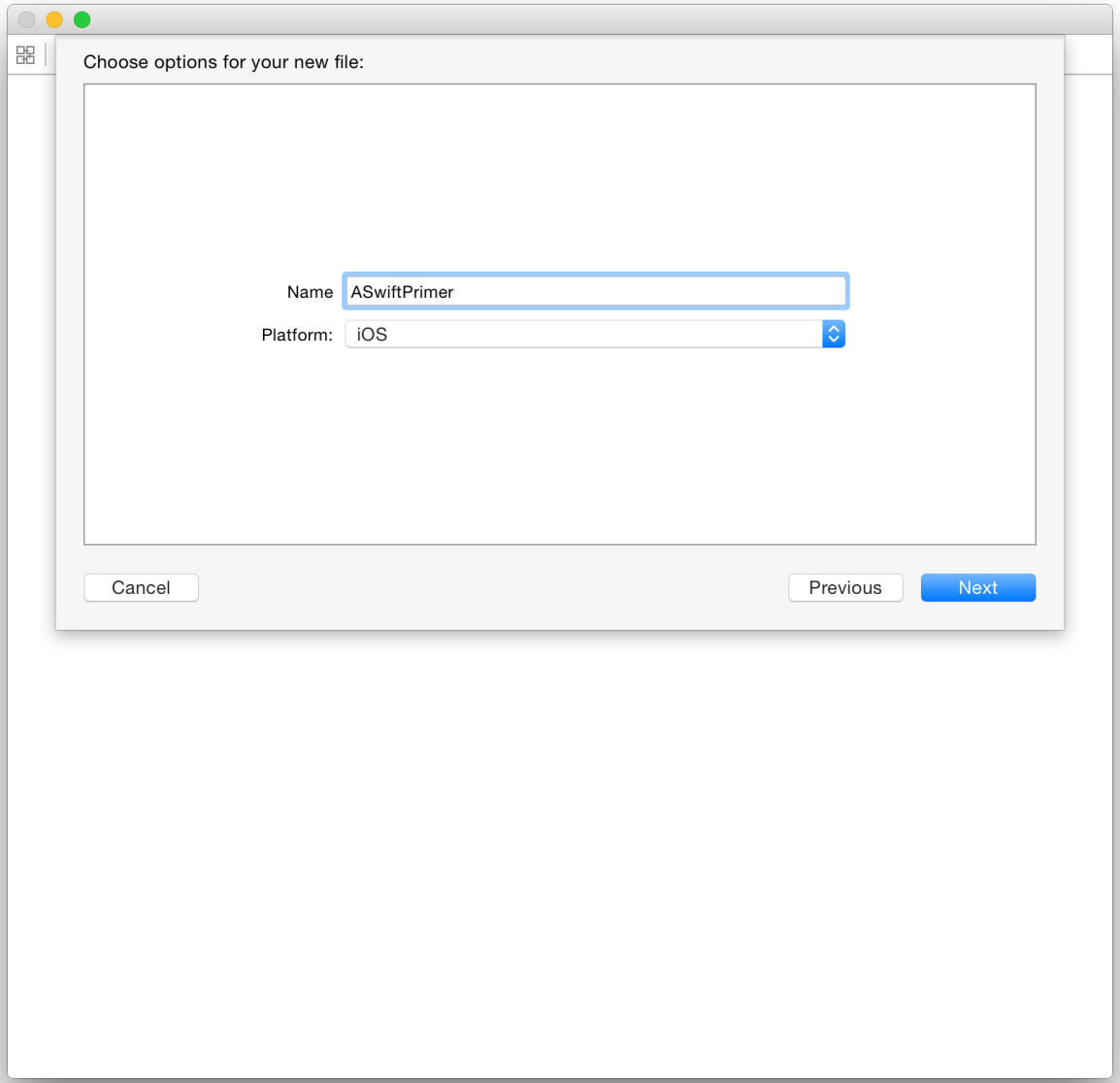
Source: https://developer.apple.com/swift/resources/

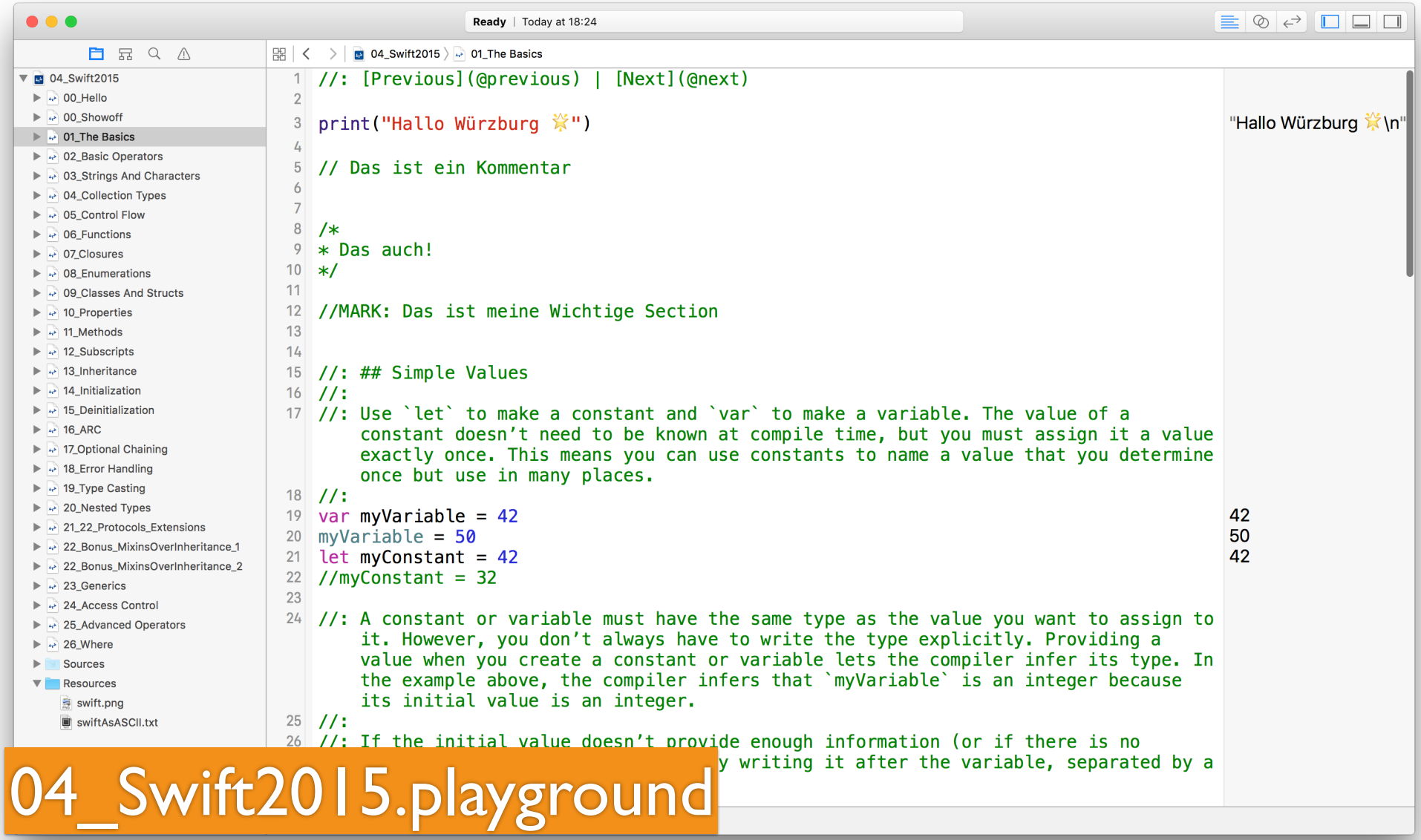
Playgrounds (2/2)

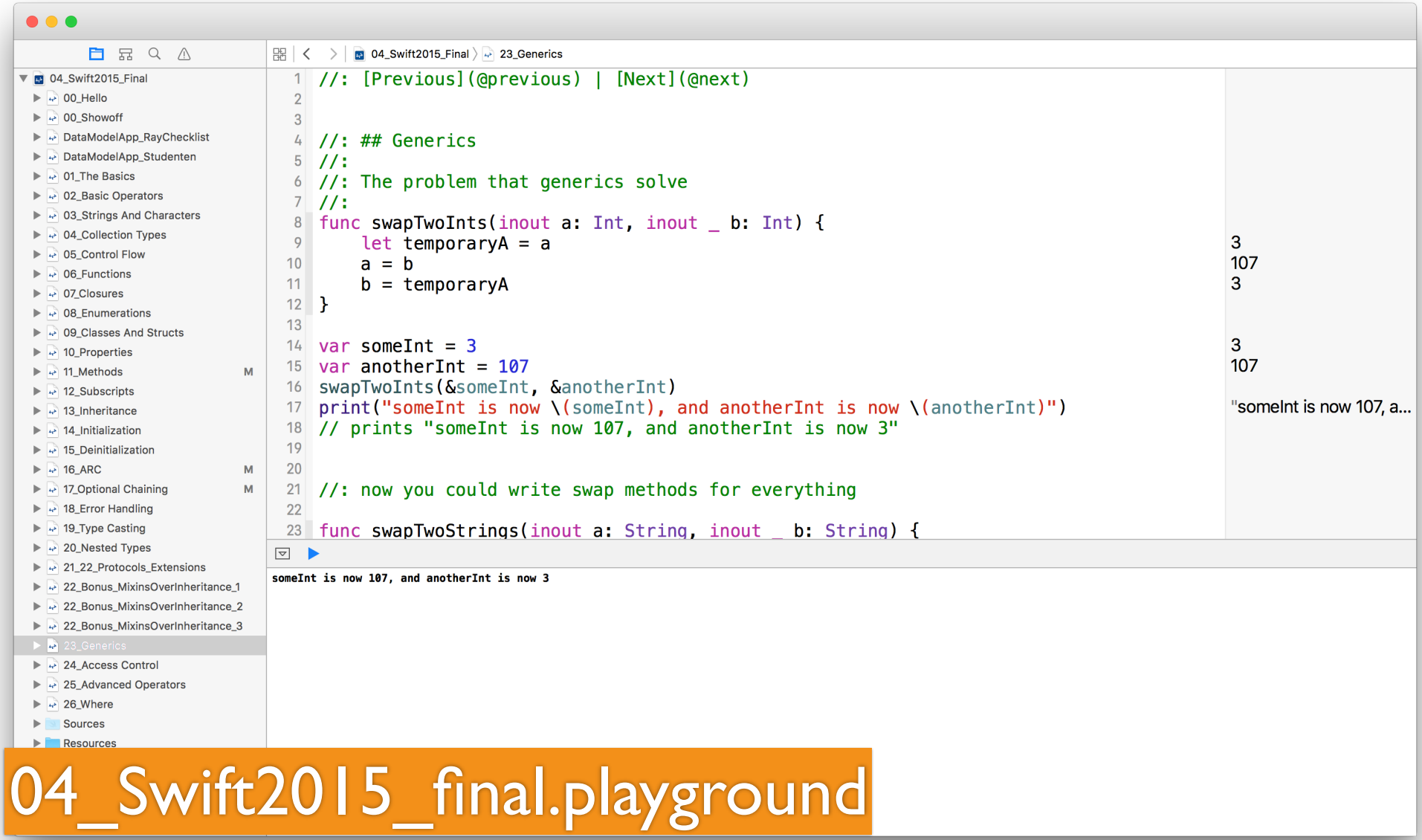


Source: http://www.objc.io/issue-16/rapid-prototyping-in-swift-playgrounds.html


BETA
XCODE
Welcome to Xcode
Version 6.2 (6C101)
Get started with a playground
Explore new ideas quickly and easily.
Create a new Xcode project
Start building a new iPhone, iPad or Mac application.
Check out an existing project
Start working on something from an SCM repository.
No Recent Projects
Open another project…


Choose options for your new file:
Name
ASwiftPrimer
Platform:
iOS
Cancel
Previous
Next

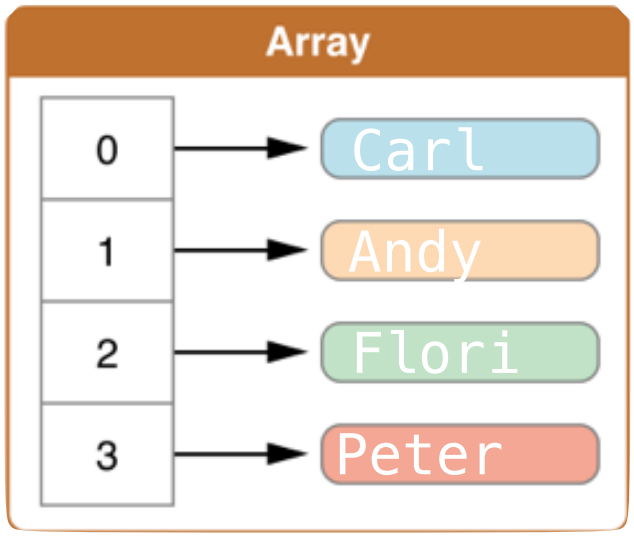

Ready | Today at 18:24
04_Swift2015
01_The Basics
04_Swift2015
00_Hello
00_Showoff
01_The Basics
02_Basic Operators
03_Strings And Characters
04_Collection Types
05_Control Flow
06_Functions
07_Closures
08_Enumerations
09_Classes And Structs
10_Properties
11_Methods
12_Subscripts
13_Inheritance
14_Initialization
15_Deinitialization
16_ARC
17_Optional Chaining
18_Error Handling
19_Type Casting
20_Nested Types
21_22_Protocols_Extensions
22_Bonus_MixinsOverInheritance_1
22_Bonus_MixinsOverInheritance_2
23_Generics
24_Access Control
25_Advanced Operators
26_Where
Sources
Resources
swift.png
swiftAsASCII.txt
//: [Previous](@previous) | [Next](@next)
print("Hallo Würzburg 🌟")
// Das ist ein Kommentar
/*
* Das auch!
*/
//MARK: Das ist meine Wichtige Section
//: ## Simple Values
//:
//: Use `let` to make a constant and `var` to make a variable. The value of a constant doesn't need to be known at compile time, but you must assign it a value exactly once. This means you can use constants to name a value that you determine once but use in many places.
//:
var myVariable = 42
myVariable = 50
let myConstant = 42
//myConstant = 32
//: A constant or variable must have the same type as the value you want to assign to it. However, you don't always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that `myVariable` is an integer because its initial value is an integer.
//:
//: If the initial value doesn't provide enough information (or if there is no
y writing it after the variable, separated by a
"Hallo Würzburg 🌟\n"
42
50
42
04_Swift2015.playground

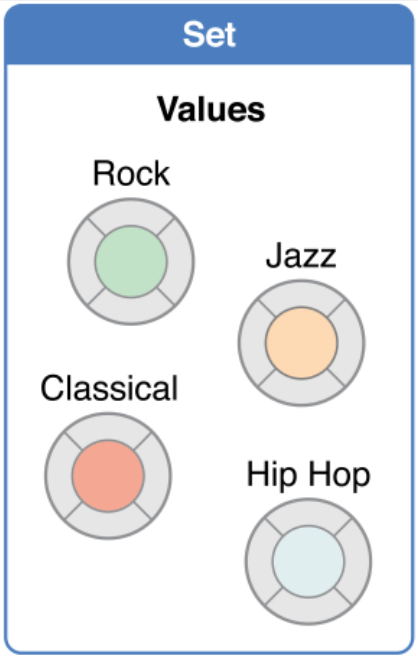


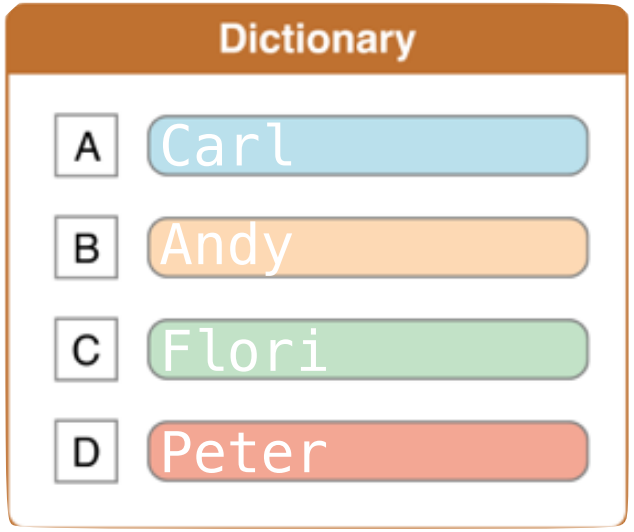
04_Swift2015_final.playground

04: Collections Reminder

```
["A": "Carl", "B": “Andy","C": "Flori", "D": "Peter"]
```



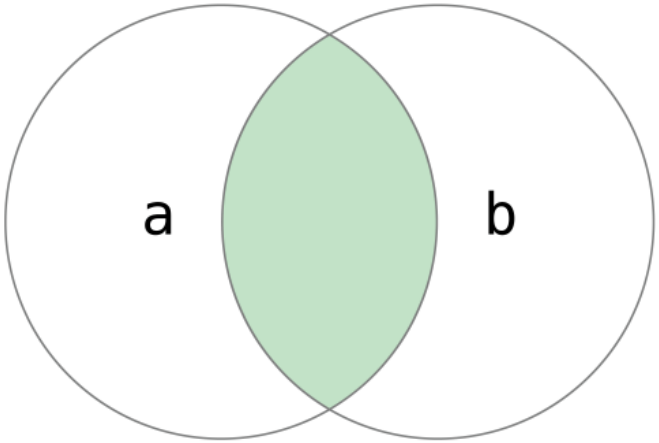




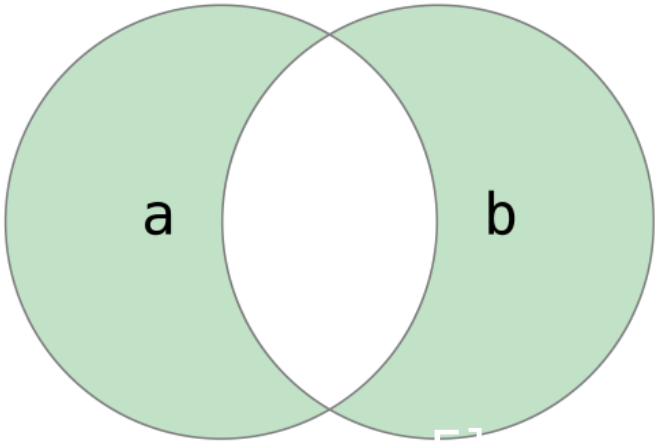

```
["Carl", "Andy", “Flori","Peter"]
```

04: Set operations

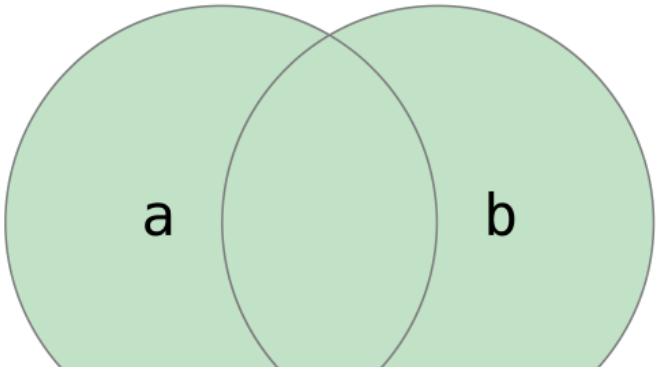
`a.intersect(b)`



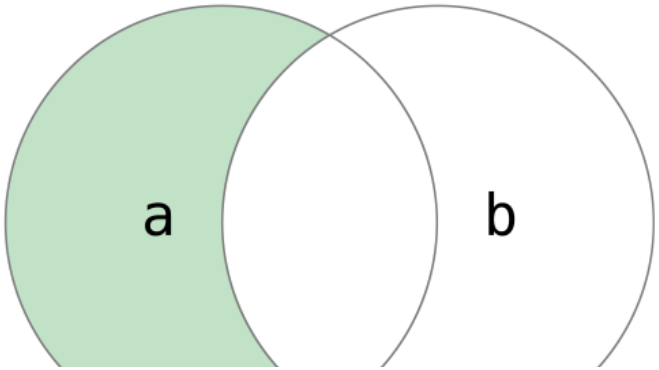
`a.exclusiveOr(b)`



`a.union(b)`



`a.subtract(b)`



04: Order of Collections

- In many languages and frameworks (including Swift) sets and dictionaries do not guarantee any kind of order, in contrast to arrays.
- An ordered dictionary is like a normal dictionary, but the keys are in a defined order.
- You'll use this functionality to store search results keyed by search term, making it quick to find results and also to maintain the order for the table view.

05: if, let, guard & where (1/3)

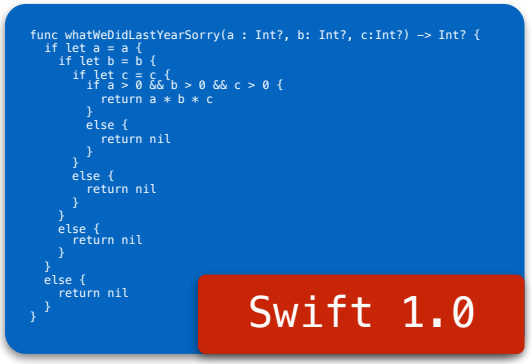
```
func whatWeDidLastYearSorry(a : Int?, b: Int?, c:Int?) -> Int? {
  if let a = a {
    if let b = b {
      if let c = c {
        if a > 0 && b > 0 && c > 0 {
          return a * b * c
        }
        else {
          return nil
        }
      }
      else {
        return nil
      }
    }
    else {
      return nil
    }
  }
  else {
    return nil
  }
}
```

Pyramid of Doom

Swift 1.0

05: if, let, guard & where (2/3)

```
func swift12MadeItBetter(a : Int?, b: Int?, c:Int?) -> Int? {
  if let a = a, b = b, c = c {
    if a > 0 && b > 0 && c > 0 {
      //do stuff
      return a * b * c
    }
    else {
      return nil
    }
 }
  else {
    return nil
  }
```



where makes it even better

```
func letAndWhere(a : Int?, b: Int?, c:Int?) -> Int? {
  if let a = a, b = b, c = c where a > 0 && b > 0 && c > 0 {
    //do stuff
    return a * b * c
  }


  return nil
}
```

Swift 1.2

05: if, let, guard & where (3/3)

```
func letAndWhere(a : Int?, b: Int?, c:Int?) -> Int? {
  if let a = a, b = b, c = c where a > 0 && b > 0 && c > 0 {
    //do stuff
    return a * b * c
  }


  return nil
}
```

```
func nowWithGuard(a : Int?, b: Int?, c:Int?) -> Int? {
  guard let a = a, b = b, c = c where a > 0 && b > 0 && c > 0 else {
    return nil
  }

  //do stuff
  return a * b * c
}
```

Swift 2.0

07: Closures (1/2)

or “unnamed functions”

```
let age = 28
let name = “Carl”

receiver.goClosuresStuff( closure )
```

Sender

“I’m sending a message and passing it a block of code.”

Receiver

“I still have access to name and age, even though I am invoked over here!”

07: Closures (2/2)

or “unnamed functions”

```
let age = 28
let name = “Carl”

receiver.goClosuresStuff({
  print(age)
  print(name)
})
```

Sender

“I’m sending a message and passing it a block of code.”

Receiver

“I still have access to name and age, even though I am invoked over here!”

08: Enums - Variable Names (1/4)

Names, datatypes and meaning

```
var person = 2

if person == 0 {
  println(“Carl”)
} else if person == 1 {
  println (“Andy”)
} else …
```

Is persons really about Numbers?

08: Enums - Variable Names (2/4)

Names, datatypes and meaning

```
let personCarl = 1
let personAndy  = 2
let personFlori = 3
let personPeter = 4

var person =  personCarl

if person == personCarl {
 println(“Carl”)
} else if person ==
personName2{
 println (“Andy”)
}…
```

Using static variables with a clearer naming

but person is still an Int.

08: Enums - Variable Names (3/4)

Names, datatypes and meaning

```
enum Person{
    case Carl
    case Andy
    case Flori
    case Peter
}

var person =  Person.Carl
```

Using Enumerations to handle different cases

08: Enums - Variable Names (4/4)

Names, datatypes and meaning

```
enum Person{
    case Carl
    case Andy
    case Flori
    case Peter
}

var person =  Person.Carl

person = .Andy
```

Or even better, the short version :-)

09: Classes & Structs

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

Fun Fact: Structs live on stack

https://github.com/nettlep/learn-swift

09: Class only (not for Structs)

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

Fun Fact: Classes live on heap

09: Struct vs Class

Structs (, Enums & Tuples)

- value type
- short lived objects
- objects that are created often
- model objects
- data capsules

Classes

- reference type
- long lived objects
- controller and view objects
- class hierarchies (inheritance (watch for final))
- objects in the true sense (representing some identity)

If unsure, try a struct first; you can change it later

Source: RWTH-Aachen: iOS WS14/15

09: Classes & Structs

Fun Fact:

- The public types in Swift's standard library:
- 87% are structs
- 8% enums
- 5% classes
- (maybe it is because it is the standard lib, but it gives a hint, how important structs are)

09: Choosing Between Classes and Structures (1/3)

- You can use both classes and structures to define custom data types to use as the building blocks of your program's code.
- structure instances are always passed by value
- class instances are always passed by reference
- This means that they are suited to different kinds of tasks. As you consider the data constructs and functionality that you need for a project, decide whether each data construct should be defined as a class or as a structure.

09: Choosing Between Classes and Structures (2/3)

Choose a struct when one or more of these conditions apply:

- The structure's primary purpose is to encapsulate a few relatively simple data values.
- It is reasonable to expect that the encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure.
- Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
- The structure does not need to inherit properties or behavior from another existing type.

09: Choosing Between Classes and Structures (3/3)

Examples of good candidates for structures include:

- The size of a geometric shape, perhaps encapsulating a width property and a height property, both of type Double.
- A way to refer to ranges within a series, perhaps encapsulating a start property and a length property, both of type Int.
- A point in a 3D coordinate system, perhaps encapsulating x, y and z properties, each of type Double.

```
for everything else
class
```

10: Common Patterns - Singleton

- It is a singleton! :-)
- Please do not try to overuse it
- it is a bit like heroin! Easy to use, hard to get rid of

```
class SomeManagerClass {
  static let sharedInstance = SomeManagerClass()
  private init() {} //This prevents others from using
                    //the default '()' initializer
                    //for this class.
  // More class code here
}
```

```
let singleton = SomeManagerClass.sharedInstance
singleton.doSomething()
```

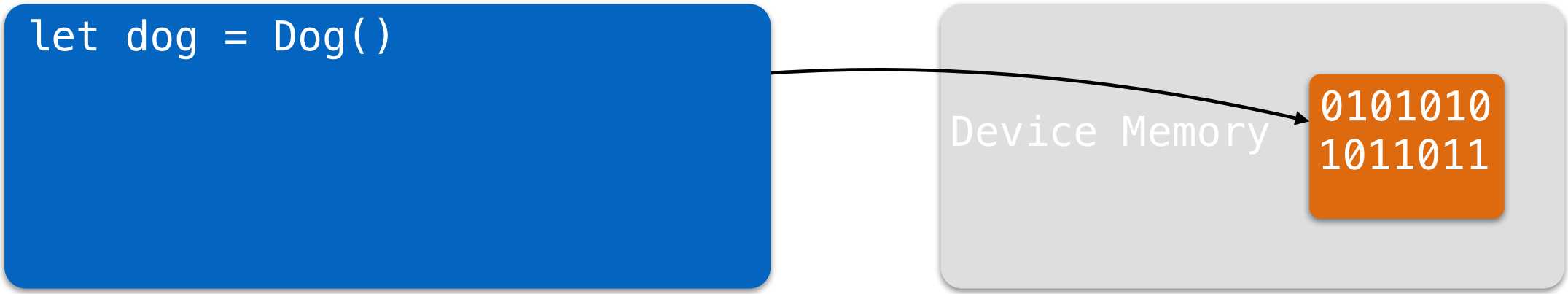
10: Other Example for `static`

```
struct AudioChannel {
  static let thresholdLevel = 10
  static var maxInputLevel = 0
  var currentLevel: Int = 0 {
    didSet {
      if currentLevel > AudioChannel.thresholdLevel {
        // cap the new audio level to the threshold level
        currentLevel = AudioChannel.thresholdLevel
      }
      if currentLevel > AudioChannel.maxInputLevel {
        // store this as the new overall maximum input level
        AudioChannel.maxInputLevel = currentLevel
      }
    }
  }
}
```


Source: Apple - The Swift Programming Language: Properties

14: Initialization (1/5)

Creating Objects



14: Initialization (2/5)

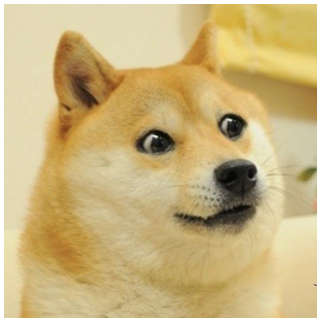
Creating Objects

```
class Dog {
    let size = “small”
    let name = “No name”

    func bark() {
      print(“wuff”)
    }
}
```

```
let dog = Dog()
```

My name is “No name”
and my size is “small”



14: Initialization (3/5)

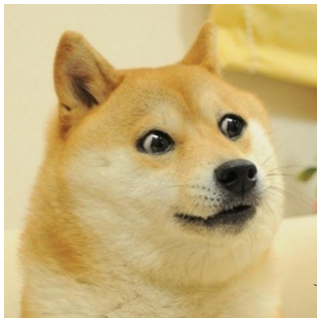
Creating Objects

```
class Dog {
   var size:String
   var name:String

   init(name:String,size:String){
     self.name = name
     self.size = size
   }
   func bark() {
     print(“wuff”)
   }
}
```

```
let dog = Dog(“Dog”,”Small”)
```

My name is “Dog”
and my size is “Small”



14: Initialization (4/5)

Creating Objects

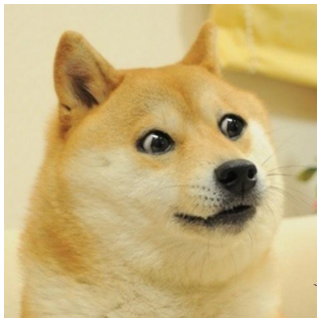
```
class Dog {
   var size:String
   var name:String

   init(){
     name = “No Name”
     size = “Small”
   }
   init(name:String) {
     self.name = name
     self.size = “small”
   }
   init(name:String,size:String) {
     self.name = name
     self.size = size
   }
   func bark() {
```

```
let dog = Dog(“Dog”)
```

My name is “Dog”

and my size is “Small”



14: Initialization (5/5)

Creating Objects

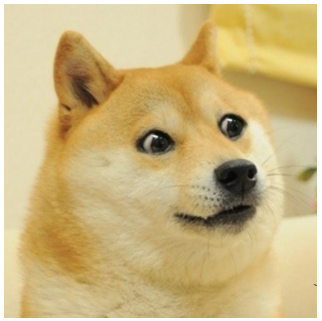
```
class Dog {
   var size:String
   var name:String

   convenience init(){
     self.init(name:”Dog”, size:”small”)
   }
   convenience init(name:String) {
     self.init(name:name, size:”small”)
   }
   init(name:String,size:String) {
     self.name = name
     self.size = size
   }
   func bark() {
     print(“wuff”)
   }
```

```
let dog = Dog(“Dog”)
```

My name is “Dog”

and my size is “Small”



16: ARC (1/22)

- ARC (Automatic Reference Counting) allows Swift to track and manage your app's memory usage.
- It automatically frees up memory from unused instances that are no longer in use.
- Reference counting only applies to classes as structures and enumerations are value types.
- Whenever a class instance is stored (to a property, constant or variable) a "**strong reference**" is made. A strong reference ensures that the reference is not deallocated for as long as the strong reference remains.

16: ARC (2/22)

```
class Person {
 let name: String
 init (name: String) {
  self.name = name
 }
}
```

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

ARC (3/22)

```
class Person {
 let name: String
 init (name: String) {
  self.name = name
 }
```

{{name “Andy”}}

```
var person: Person? = Person(name: "Andy")
```

one instances now

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (4/22)

```
class Person {
 let name: String
 init (name: String) {
  self.name = name
 }
```

{{name “Andy”}}

```
var person: Person? = Person(name: "Andy")
var copyOfPerson = person
```

Two instances now

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (5/22)

```
class Person {
 let name: String
 init (name: String) {
  self.name = name
 }
```

{{name “Andy”}}

```
var person: Person? = Person(name: "Andy")
var copyOfPerson = person

person = nil
```

Two instances now

one instances now

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (6/22)

```
class Person {
 let name: String
 init (name: String) {
  self.name = name
 }
```

{{name “Andy”}}

```
var person: Person? = Person(name: "Andy")
var copyOfPerson = person

person = nil

copyOfPerson = nil
```

Two instances now

one instances now

zero instances now

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (7/22)

```
class Person {
 let name: String
 init (name: String) {
  self.name = name
 }
```

{{name “Andy”}}

```
var person: Person? = Person(name: "Andy")
var copyOfPerson = person

person = nil

copyOfPerson = nil
```

Two instances now

one instances now

zero instances now

release :-)

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (8/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (9/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (10/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (11/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

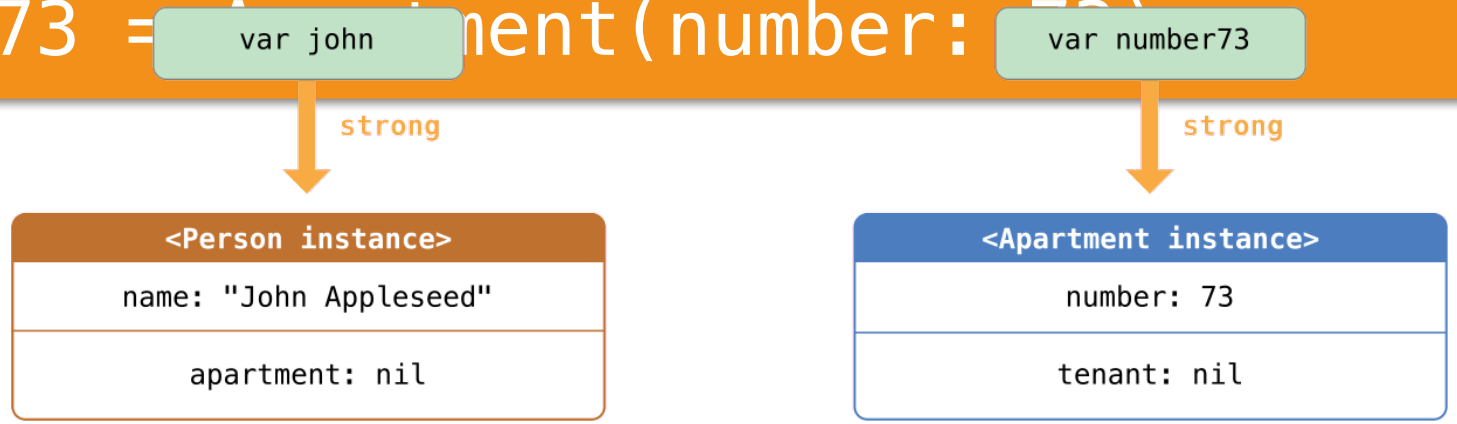
16: ARC (12/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

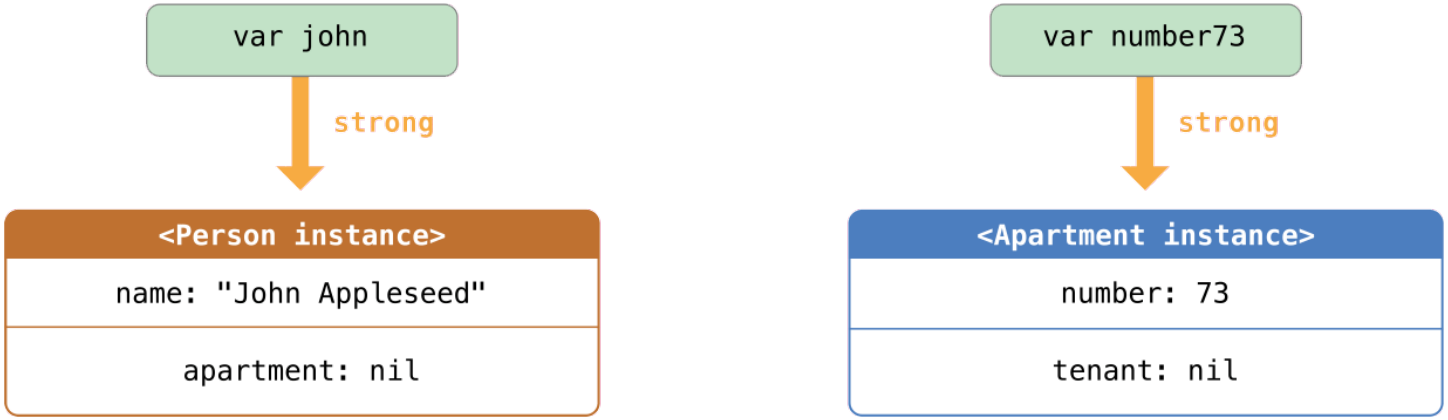
16: ARC (13/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (14/22) - Cycles

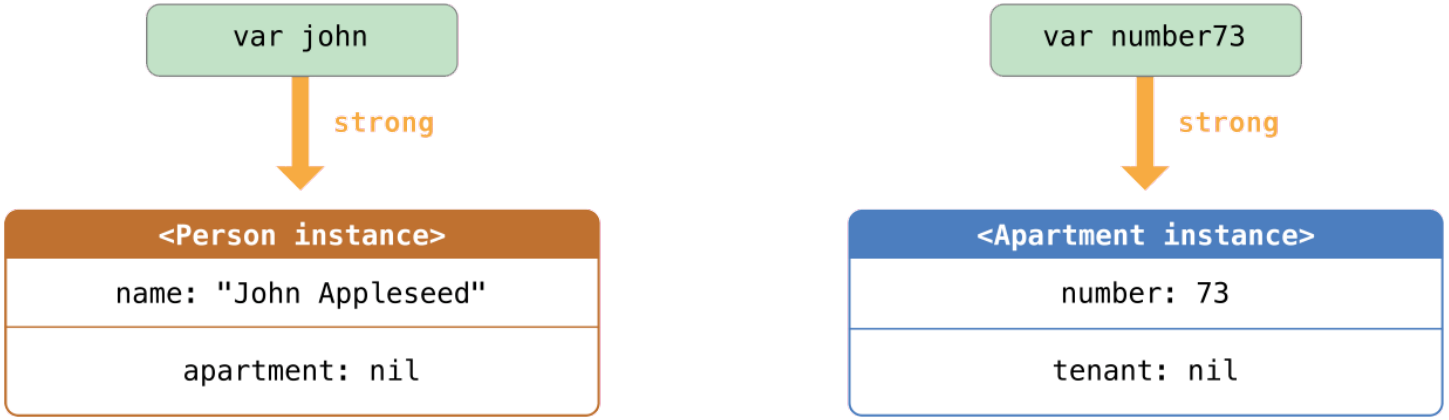
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (15/22) - Cycles

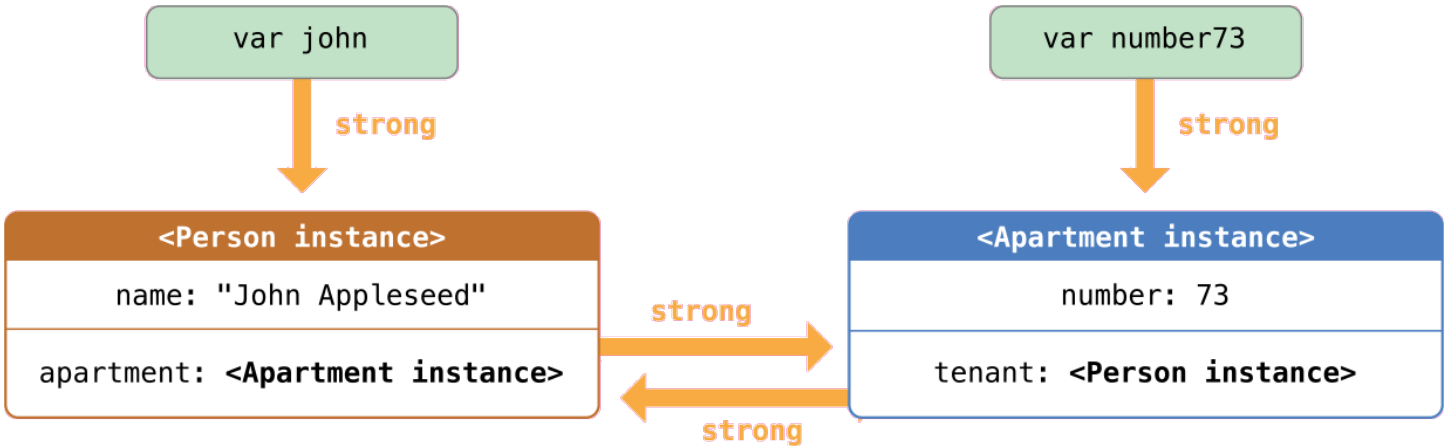
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (16/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

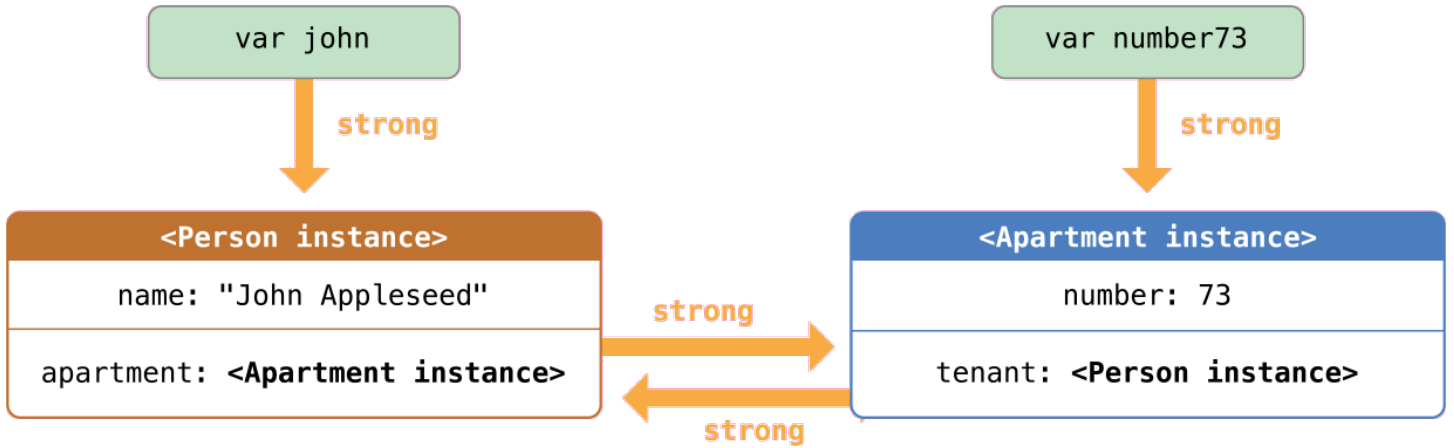
```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```

```
john = nil
number73 = nil
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (17/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

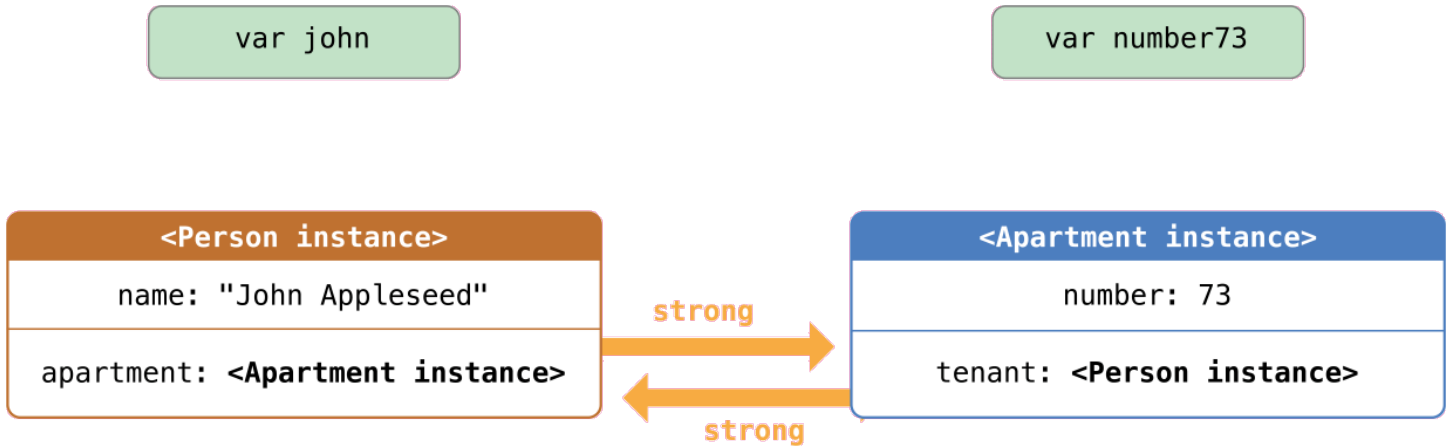
```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```

```
john = nil
number73 = nil
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (18/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
}
```

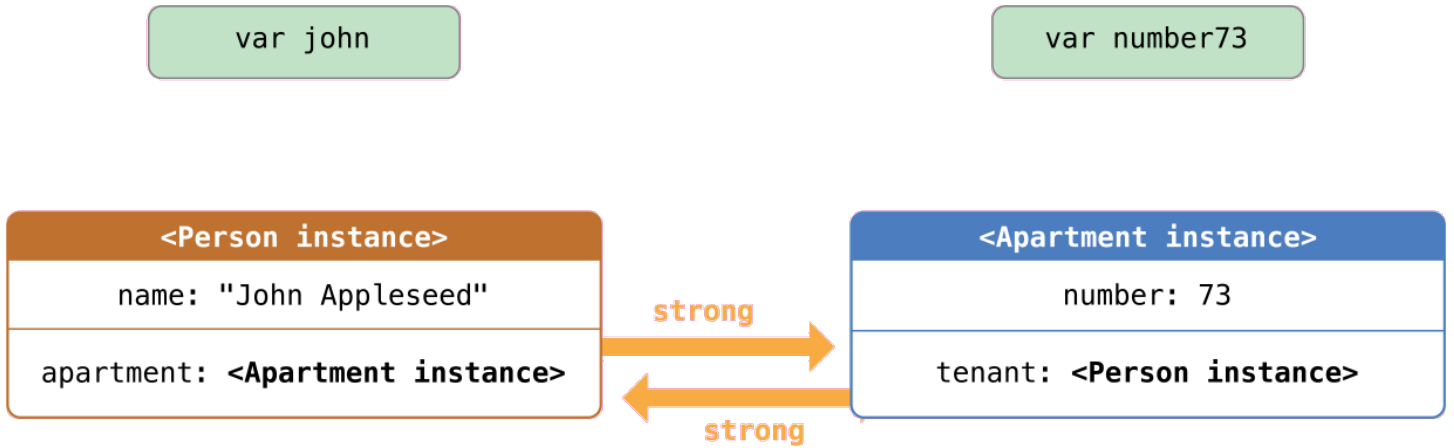
```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```

The strong references between the Person instance and the Apartment instance remain and cannot be broken.

```
john = nil
number73 = nil
```



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (19/22) - Weak

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

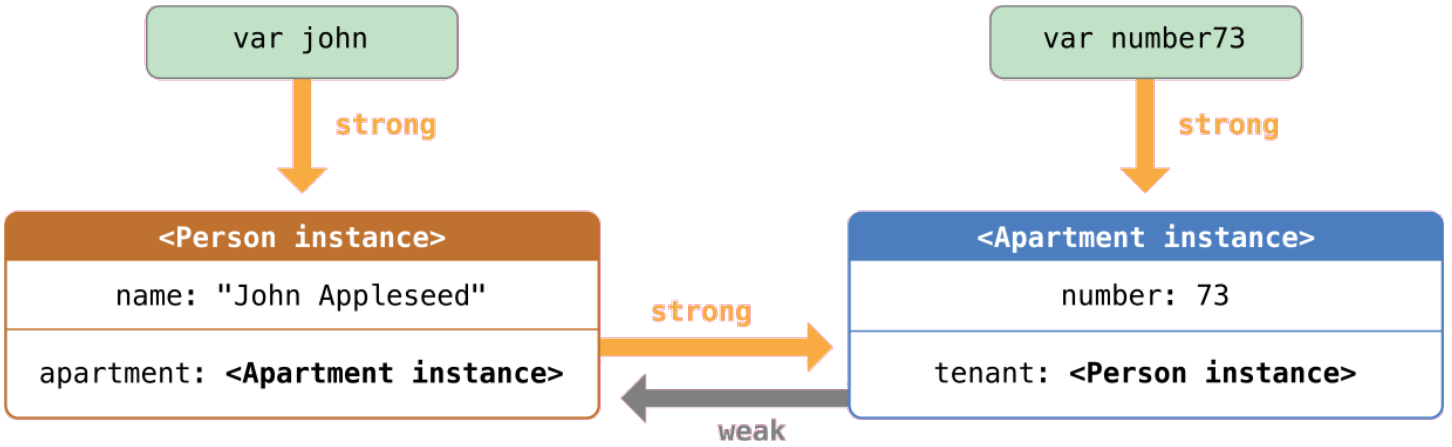
```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    weak var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```

WEAK to the rescue!



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (20/22) - Weak

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    weak var tenant: Person?
}
```

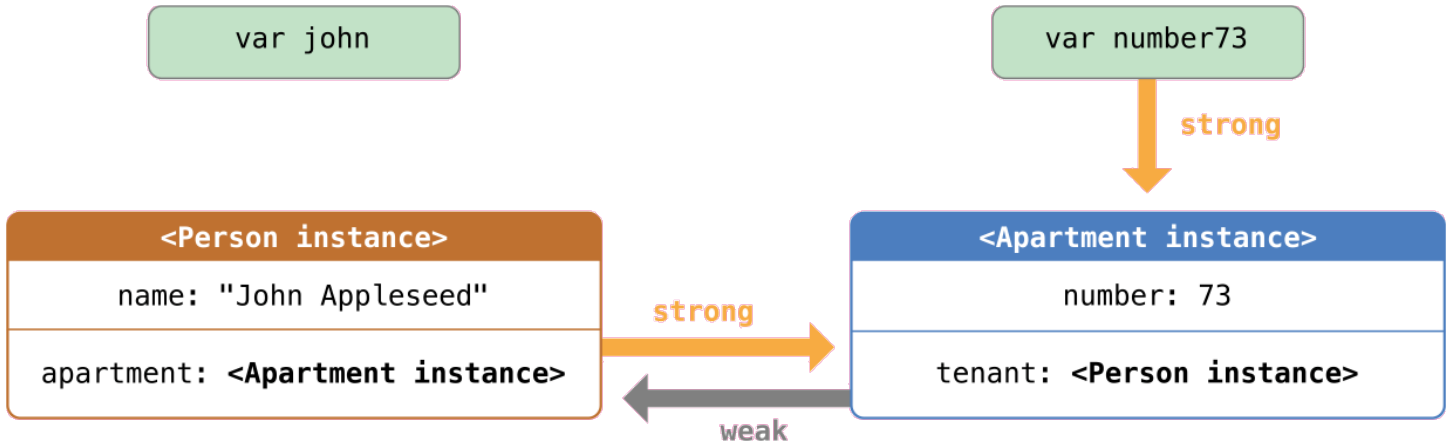
```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```

```
john = nil
```

WEAK to the rescue!



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (21/22) - Weak

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    weak var tenant: Person?
}
```

```
var john: Person?
var number73: Apartment?
```

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

```
john!.apartment = number73
number73!.tenant = john
```

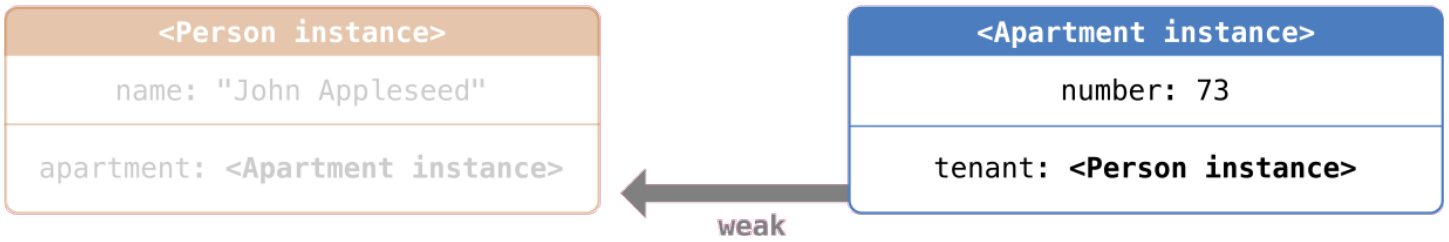
WEAK to the rescue!

```
john = nil
```

```
nubmer73 = nil
```

n

var number73



Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

16: ARC (22/22) - Cycles

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}
```

```
class Apartment {
    let num: Int
    init(num: Int) { self.num = num }
    var tenant: Person?
```

```
var john: Person?
var number73: Apartment?
```

```
john =
number7
```

```
john!.a
number7
```

Avoid strong reference cycles! Use weak!

```
apartment: nil
```

```
tenant: nil
```

Source: https://github.com/iwasrobbed/Swift-CheatSheet#commenting & https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

17: Optional Chaining

- is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil.
- If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil.
- can be chained together

```
 person?.musicPreferences
person?.musicPreferences?.favoriteSong?.artist

if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
}
```

19: Type casting (1/3)

- Sometimes it is necessary to cast an object into a specific class or data type. Examples of this would be casting from a Float to an Int or from a UITableViewCell to a subclass such as AwesomeTableViewCell.

`is` & `as`

19: Type casting is (2/3)

- Operator `is`

- The is operator returns **true** if an instance is of that object type
- and returns **false** if it does not.

```
if item is Movie {
    ++movieCount
    println("It is a movie.")
} else if item is Song {
    ++songCount
    println("It is a song.")
}
```

19: Type casting as 3/3

- Operator `as`

- If you want to be able to easily access the data during one of these checks, you can use as? to optionally (or as to force) unwrap the object when necessary:

```
for item in library {
    if let movie = item as? Movie {
        println("Director: \(movie.director)")
    } else if let song = item as? Song {
        println("Artist: \(song.artist)")
    }
}
```

19: Casting from Generic Types

- If you're working with AnyObject objects given from the Cocoa API, you can use:

```
for movie in someObjects as! [Movie] {
    // do stuff
}
```

21: Protocols (1/4)

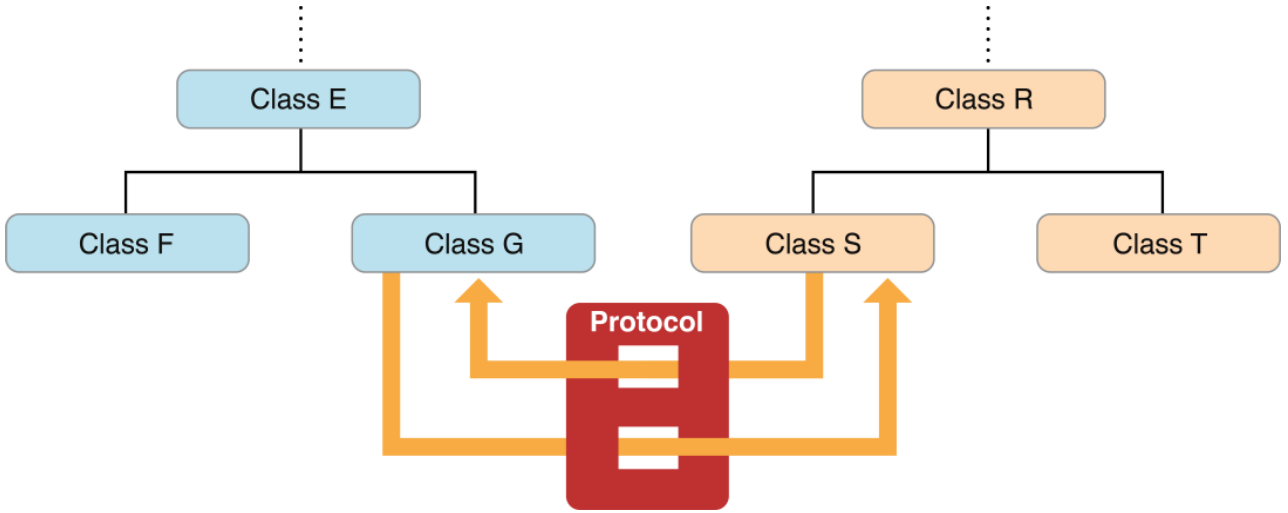
- Protocols define a required set of functionality (including methods and properties) for a class, structure or enumeration.
- Protocols are "adopted" by a class, structure or enumeration to provide the implementation.
  This is called "conforming" to a protocol.
- As you should already be aware, protocols are often used for delegation.

```
protocol FirstProtocol {
    // protocol definition goes here
}
```

```
class SomeClass: SomeSuperclass, FirstProtocol {
    // class definition goes here
}
```

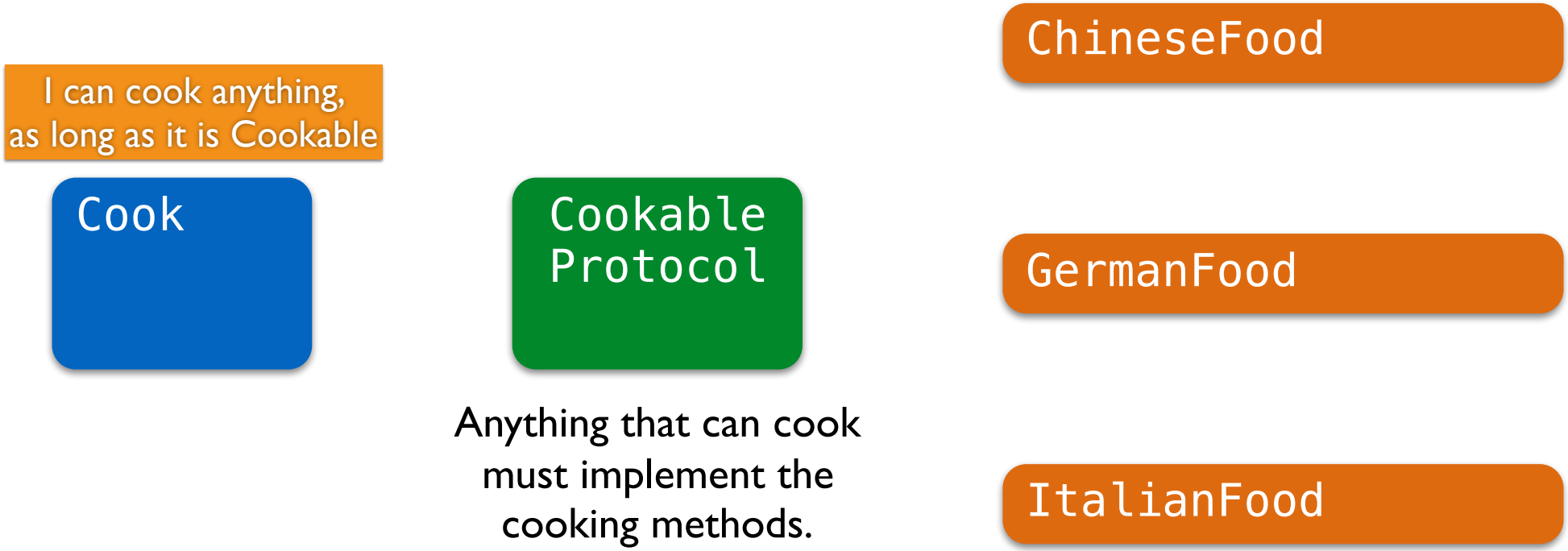
21: Protocols (2/4)

- Protocols Define Messaging Contracts
- Using protocols, two classes can communicate with each other to interchange data (or whatever is necessary)



Source: Apple - Start Developing iOS Apps Today

21: Protocols (3/4)



21: Protocols (4/4)

Cook

```
class ChineseFood : Cookable {
}
```

```
class GermanFood : Cookable {
}
```

```
class ItalianFood : Cookable {
}
```

Cookable Protocol

```
protocol Cookable {
  func cooking()
}
```

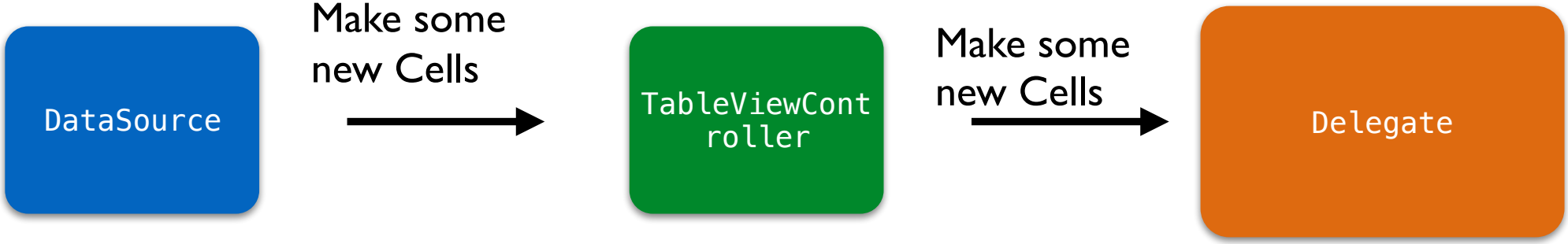
Classes which implements the Cookable promise to implement the cooking function.

21: Delegates (1/5)

"I'm just a regular chef, I don't know how to make sushi."

21: Delegates (2/5)

Hand of functionalities to another object



“I'm just a ViewController I don’t know how to make new Cells.”

21: Delegates (3/5)

Hand of functionalities to another object

I know an Object which implements the delegation

21: Delegates (4/5)

1. Create a delegate protocol that defines the responsibilities of the delegate.
2. Create a delegate property in the delegating class to keep track of the delegate.
3. Adopt and implement the delegate methods in the delegate class.
4. Call the delegate from the delegating object.

Swift Anomaly

```
 Objective-C supports optional protocol methods.
Swift only allows optional protocol requirements if you mark the protocol
with the @objc attribute.
If you use @objc you can then only use the protocol with class types.
```

21: Delegates (5/5)

1. The protocol

```
protocol CookableDelegate: class {
  func cooking()
}
```

2a. the delegate property

```
weak var delegate:CookableDelegate?
```

2b. assign

```
detailViewController.delegate = self
```

3. Adopt and implement the delegate methods

```
extension AsianFood: CookableDelegate {
  func cooking() {
    // do stuff like asian cooking
  }
}
```

4. call the delegate from the delegating object

```
delegate?.cooking(self)
```

21 & 22: Mixins over Inheritance (1/3)

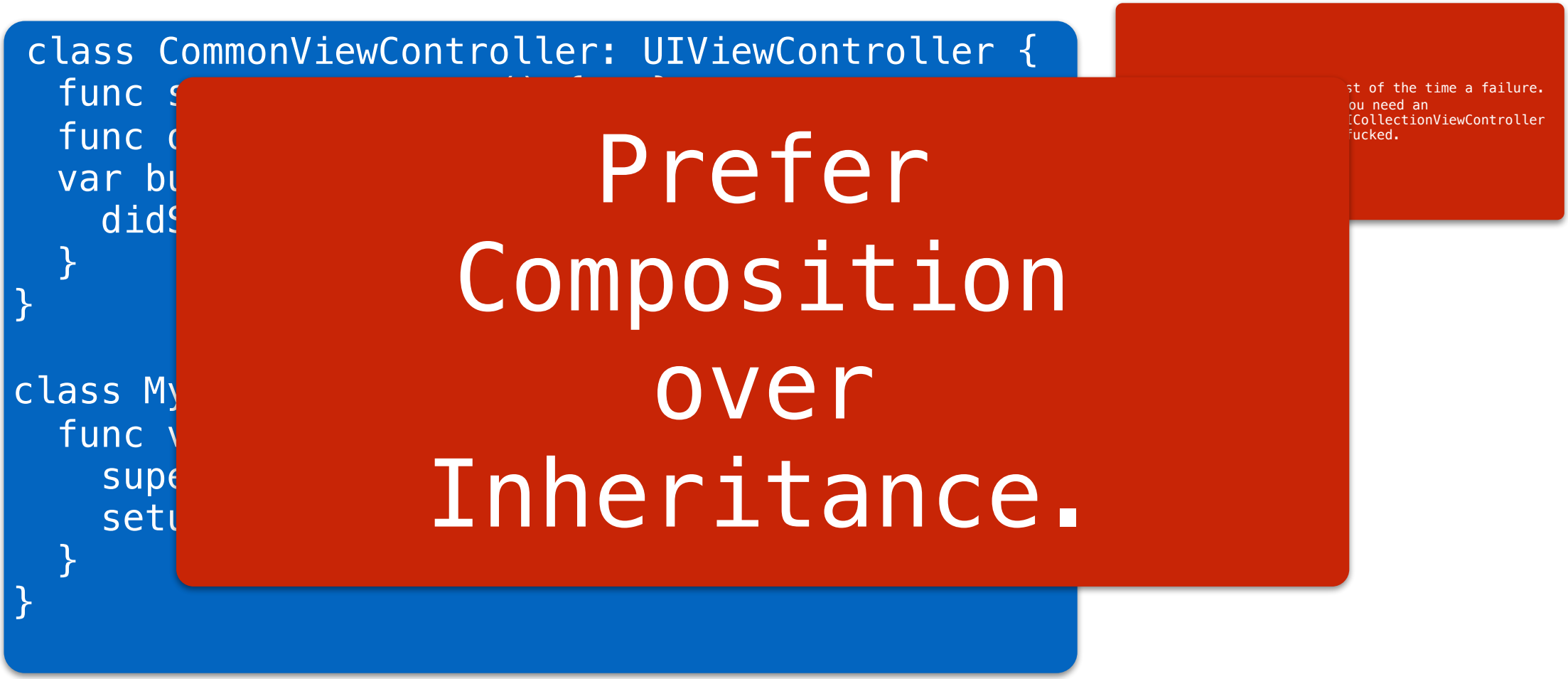
```
class CommonViewController: UIViewController {
  func setu
  func onBu
  var burge
    didSet
  }
}

class MyVie
  func viewD
    super.v
    setupBu
  }
}
```

CommonViewController is most of the time a failure. Because if you need an UITableViewController or UICollectionViewController you are fucked.

Source: Olivier Halligon: Mixins over Inheritance

21 & 22: Mixins over Inheritance (2/3)



Source: Olivier Halligon: Mixins over Inheritance

21 & 22: Mixins over Inheritance (3/3)

```
class BurgerMenuManager {
  func setupBurgerMenu() { … }
  func onBurgerMenuTapped() { … }
  func burgerMenuIsOpen: Bool { didSet { … } }
}
class MyViewController: UIViewController {
  var menuManager: BurgerMenuManager()
  func viewDidLoad() {
    super.viewDidLoad()
    menuManager.setupBurgerMenu()
  }
}
class MyOtherViewController: UITableViewController {
  var menuManager: BurgerMenuManager()
  func viewDidLoad() {
    super.viewDidLoad()
    menuManager.setupBurgerMenu()
  }
}
```

CommonViewController is most of the time a failure. Because if you need an UITableViewController or UICollectionViewController you are fucked.

Prefer Composition over Inheritance.

Source: Olivier Halligon: Mixins over Inheritance

26: Swift Standard Library (1/2)

The Swift standard library implements the basic data types, algorithms, and protocols you use to write apps in Swift. It includes high-performance fundamental data types such as `String` and `Array`, along with generic algorithms such as `sort()` and `filter()`. Powerful protocols that describe shared traits and behaviors define reusable building blocks for higher level types. Classes in the Foundation framework with standard library counterparts–such as `NSArray` and `Array`–are automatically bridged. When you work with Objective-C frameworks, your Swift code gains the performance improvements and flexibility of the Swift standard library.

On each page of this playground, you will experiment with Swift standard library types and high-level concepts using visualizations and practical examples. You will also learn how the Swift standard library uses protocols and generics to express powerful constraints.

https://developer.apple.com/library/prerelease/ios/documentation/General/Reference/SwiftStandardLibraryReference/

26: Swift Standard Library (2/2)

abs(_:)
alignof(_:)
alignofValue(_:)
anyGenerator<G : GeneratorType>(_: G) -> AnyGenerator<G.Element>
anyGenerator<Element>(_: () -> Element?) -> AnyGenerator<Element>
assert(_:_:file:line:)
assertionFailure(_:file:line:)
debugPrint(_:separator:terminator:)
debugPrint(_:separator:terminator:toStream:)
dump(_:_:name:indent:maxDepth:maxItems:)
dump(_:name:indent:maxDepth:maxItems:)
fatalError(_:file:line:)
getVaList(_:)
isUniquelyReferenced(_:)
isUniquelyReferencedNonObjC<T : AnyObject>(_: T?) -> Bool
isUniquelyReferencedNonObjC<T : AnyObject>(_: T) -> Bool
max(_:_:)
max(_:_:_:_:)
min(_:_:)
min(_:_:_:_:)
numericCast<T : UnsignedIntegerType, U : UnsignedIntegerType>(_: T) -> U
numericCast<T : UnsignedIntegerType, U : _SignedIntegerType>(_: T) -> U
numericCast<T : _SignedIntegerType, U : _SignedIntegerType>(_: T) -> U
numericCast<T : _SignedIntegerType, U : UnsignedIntegerType>(_: T) -> U
precondition(_:_:file:line:)
preconditionFailure(_:file:line:)
print(_:separator:terminator:)
print(_:separator:terminator:toStream:)
readLine(stripNewline:)
sizeof(_:)
sizeofValue(_:)
strideof(_:)
strideofValue(_:)
swap(_:_:)
transcode(_:_:_:_:stopOnError:)
unsafeAddressOf(_:)
unsafeBitCast(_:_:)
unsafeDowncast(_:)
unsafeUnwrap(_:)
withExtendedLifetime<T, Result>(_: T, _: T throws -> Result) rethrows -> Result
withExtendedLifetime<T, Result>(_: T, _: () throws -> Result) rethrows -> Result
withUnsafeMutablePointer(_:_:)
withUnsafeMutablePointers(_:_:_:)
withUnsafeMutablePointers(_:_:_:_:)
withUnsafePointer(_:_:)
withUnsafePointers(_:_:_:)
withUnsafePointers(_:_:_:_:)
withVaList<R>(_: [CVarArgType], _: CVaListPointer -> R) -> R
withVaList<R>(_: VaListBuilder, _: CVaListPointer -> R) -> R
zip(_:_:)

https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_StandardLibrary_Functions/index.html

Extra - Ternary Conditional Operator

```
question ? answer1 : answer2
```

- short for:

```
if question {
    answer1
} else {
    answer2
}
```

- Example:

```
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
```

Extra - Nil Coalescing Operator

```
(a ?? b)
```

- short for:

```
a != nil ? a! : b
```

- Example:

```
let trackName = result["trackName"] as? String
let collectionName = result["collectionName"] as? String
let name : String = trackName ?? collectionName
```

```
let name = trackName ?? collectionName ?? "Untitled"
```

Optional Binding

- You can unwrap an optional in both a "safe" and "unsafe" way. The safe way is to use *Optional Binding*:

```
let possibleString : String? = "Hello"
if let actualString = possibleString {
    // actualString is a normal (non-optional) String value
    // equal to the value stored in possibleString
    print(actualString)
}
else {
    // possibleString did not hold a value, handle that
    // situation
}
```


http://www.drewag.me/posts/what-is-an-optional-in-swift#unwrapping-an-optional

Unwrapping Optionals

- Sometimes you know for sure that a variable holds an actual value and you can assert that with *Forced Unwrapping* by using an exclamation point (!):

```
 let possibleString : String? = "Hello"
print(possibleString!)
```

- An *Implicitly Unwrapped Optional* is an optional that does not need to be unwrapped because it is done implicitly. These types of optionals are declared with an ! instead of a ?:

```
 let possibleString : String!
print(possibleString)
```


http://www.drewag.me/posts/what-is-an-optional-in-swift#unwrapping-an-optional

Implicitly Unwrapped Optionals 1/4

4 use cases

1. **A Constant That Cannot Be Defined During Initialization**
2. Interacting with an Objective-C API
3. When Your App Cannot Recover From a Variable Being nil
4. NSObject Initializers

```
class MyView : UIView {
    @IBOutlet var button : UIButton
    var buttonOriginalWidth : CGFloat!

    override func viewDidLoad() {
        self.buttonOriginalWidth = self.button.frame.size.width
    }
}
```

Implicitly Unwrapped Optionals 2/4

4 use cases

1. A Constant That Cannot Be Defined During Initialization
2. **Interacting with an Objective-C API**
3. When Your App Cannot Recover From a Variable Being nil
4. NSObject Initializers

```
 //MARK: UITableViewDataSource
override func tableView(tableView: UITableView!,
cellForRowAtIndexPath indexPath: NSIndexPath!) ->
UITableViewCell? { return nil }
```

Implicitly Unwrapped Optionals 3/4

4 use cases

1. A Constant That Cannot Be Defined During Initialization
2. Interacting with an Objective-C API
3. **When Your App Cannot Recover From a Variable Being nil**
4. NSObject Initializers

Implicitly Unwrapped Optionals 4/4

4 use cases

1. A Constant That Cannot Be Defined During Initialization
2. Interacting with an Objective-C API
3. When Your App Cannot Recover From a Variable Being nil
4. **NSObject Initializers**

```
var image : UIImage? = UIImage(named: "NonExistentImage")
if image.hasValue {
    println("image exists")
}
else {
    println("image does not exist")
}
```

!Implicitly Unwrapped Optionals

2 NOT use cases

1. Lazily Calculated Member Variables
2. Everywhere Else

```
class FileSystemItem {
}

class Directory : FileSystemItem {
    @lazy var contents : [FileSystemItem] = {
        var loadedContents = [FileSystemItem]()
        // load contents and append to loadedContents
        return loadedContents
    }()
}
```

Andrew Wagner - Uses for Implicitly Unwrapped Optionals in Swift

5 Alternatives for Implicitly Unwrapped Optionals

1. Optional Chaining ?
2. Nil Coalescing ??
3. You can map optionals
4. switch
5. Comparision == and !=

Source: Apple - Start Developing iOS Apps Today

The Rule of Conditional Binding Cascades

"Unless you're mixing var and let conditional bindings, use a single if let or if var introduction. Add liberal whitespace as needed."

Instead of:

```
if let x = x, let y = y, let z = z {blah}
```

use:

```
if let x = x, y = y, z = z {blah}
```

Xcode aligns:

```
if let
    x = x,
    y = y,
    z = z {
    ...blah...
}
```

The Rule of Collection Creation

"Use explicit typing and empty collections."

Instead of:

```
var x = [String: Int]()
var y = [Double]()
var z = Set<String>()
var mySet = MyOptionSet()
```

use:

```
var x: [String: Int] = [:]
var y: [Double] = []
var z: Set<String> = []
var mySet: MyOptionSet = []
```

The Rule of isEmpty

“If you’re testing a collection’s count, you’re probably doing it wrong.”

Instead of:

```
count == 0
```

use:

```
isEmpty
```

The Rule of Pattern Matching Keywords

"When everything's a binding, unify your bindings."

Instead of:

```
if case (let x?, let y?) = myOptionalTuple {
    print(x, y)
}
```

use:

```
if case let (x?, y?) = myOptionalTuple {
    print(x, y)
}
```

The Rule of readability (1/2)

Instead of:

```
if (game.state == running || trainingMode) {}
```

write:

```
let shouldPauseGame = game.state == running || trainingMode
if shouldPauseGame {}
```

and instead of:

```
if buttonIndex == 0
```

write:

```
if buttonIndex == StartButton
```

The Rule of readability (2/2)

- Clarity at the point of use is your most important goal.
  Code is read far more than it is written.
- Clarity is more important than brevity.
  Although Swift code can be compact, it is a non-goal to enable the smallest possible code with the fewest characters.
  Brevity in Swift code, where it occurs, is a side-effect of the strong type system and features that naturally reduce boilerplate.
- Write a Documentation Comment for every method or property in Swift's dialect of Markdown.
  Ideally, the API's meaning can be understood from its signature and its one- or two-sentence summary:

https://swift.org/documentation/api-design-guidelines.html#clarity-at-the-point-of-use

The Rule of readability

Instead of:

```
if (game.state == running || trainingMode) {}
```

write:

```
let shouldPauseGame = game.state == running || trainingMode
if shouldPauseGame {}
```

and instead of:

```
if buttonIndex == 0
```

write:

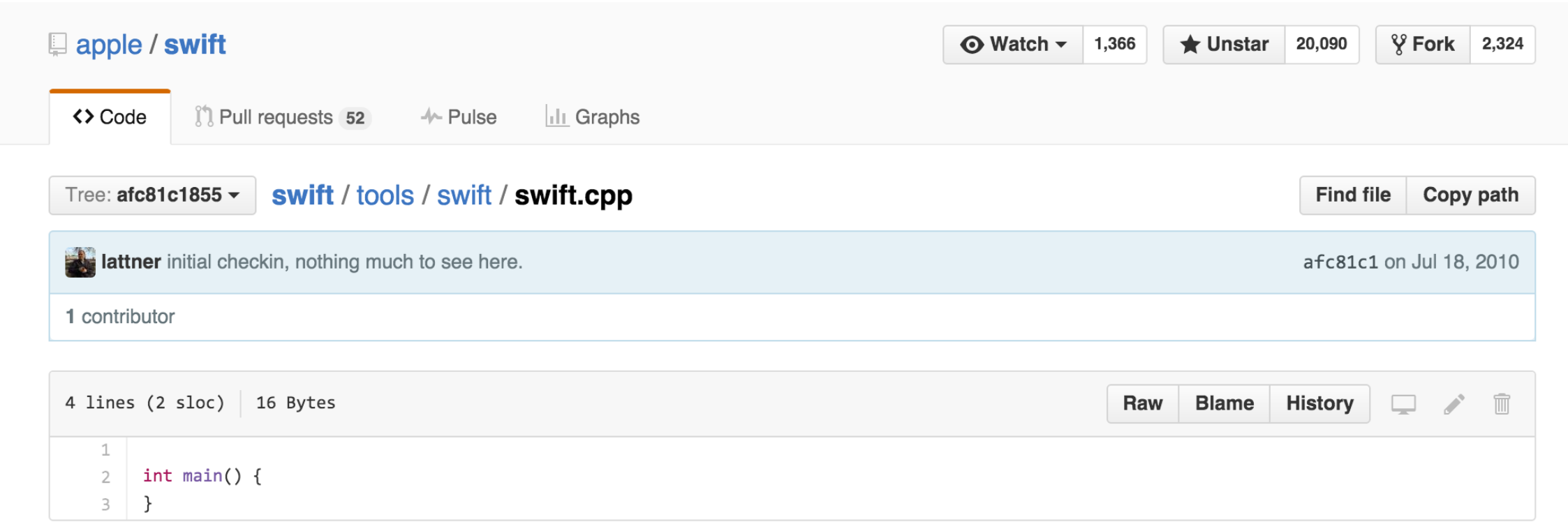
```
if buttonIndex == StartButton
```

The Rule of !

“Every time you use an exclamation point in Swift, a kitten dies.”

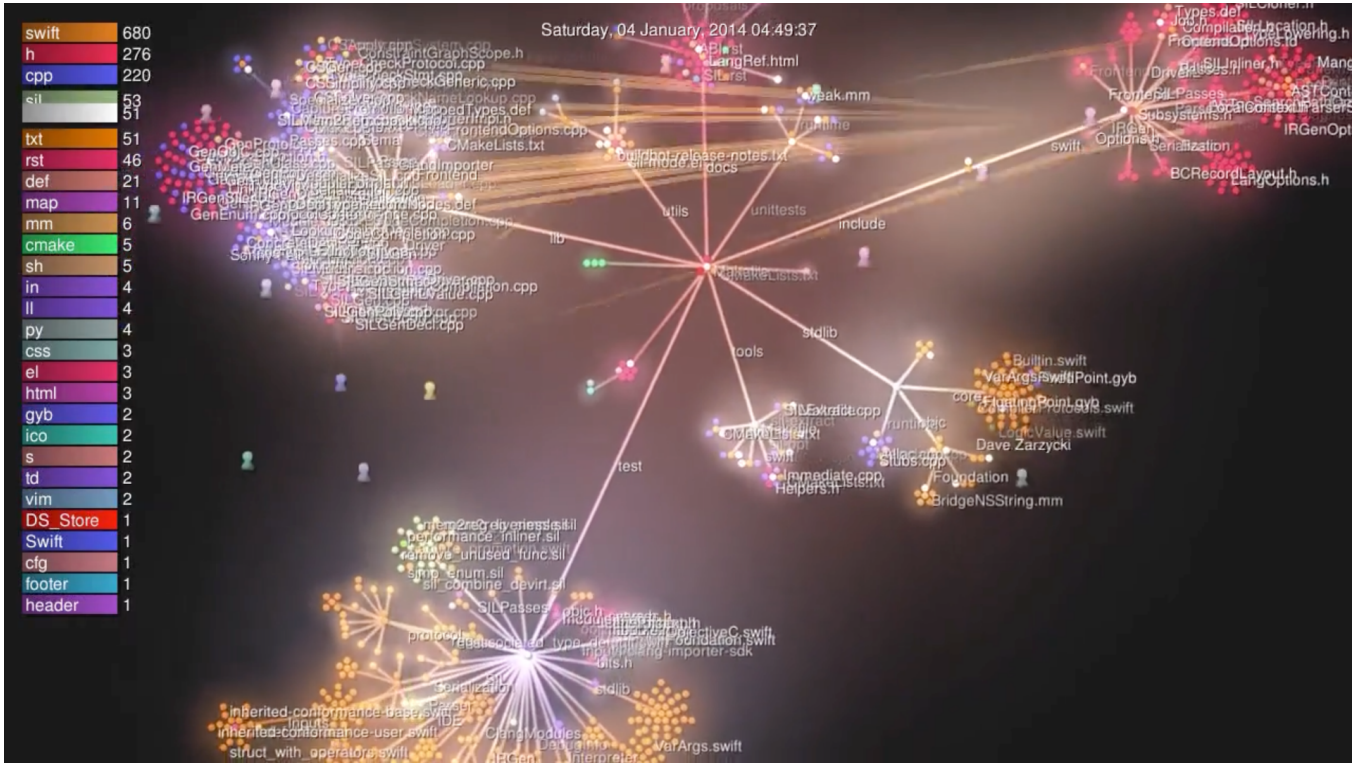
Wherever possible, avoid forced casts and forced unwrapping.

Every program starts small (please commit often)



https://github.com/apple/swift/blob/afc81c1855bf711315b8e5de02db138d3d487eeb/tools/swift/swift.cpp

The evolution of Swift



https://vimeo.com/147777653

User Parameter Objects (1/2)

Instead of:

```
func banana(length: Double, weight: Double, origin: String,
color: UIColor)
```

use:

```
struct BananaData {
    let length: Double
    let weight: Double
    let origin: String
    let color: UIColor
}

func banana(data: BananaData)
```


http://christiantietze.de/posts/2015/11/parameter-objects-simplify/

User Parameter Objects (2/2)

Instead of:

```
 func sendMoney(amount: Money, from: Account, to: Account)
func logTransaction(amount: Money, from: Account, to:
Account? = nil)
```

use:

```
struct Transfer {
    let amount: Money
    let sender: Account
    let receiver: Account?
}

func transferMoney(transfer: Transfer)
func logTransaction(transfer: Transfer)
```


http://christiantietze.de/posts/2015/11/parameter-objects-simplify/

Swift - A (not so) Quick Tour

we just have seen:

1. The Basics
2. Basic Operators
3. Strings and Characters
4. Collection Types
5. Control Flow
6. Functions
7. Closures
8. Enumerations
9. Classes and Structures
10. Properties
11. Methods
12. Subscripts
13. Inheritance
14. Initialization
15. Deinitialization
16. Automatic Reference Counting
17. Optional Chaining
18. Error Handling
19. Type Casting
20. (Nested Types)
21. Protocols
22. Extensions
23. Generics
24. (Access Control)
25. (Advanced Operators)
26. Swift Standard Library